Electronic Communications of the EASST
Volume 4 (2006)

Proceedings of the
Second International Workshop on
Graph and Model Transformation
(GraMoT 2006)

Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations

Anantha Narayanan and Gabor Karsai

14 pages

Guest Editors: Gabor Karsai, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations

Anantha Narayanan[1] **and Gabor Karsai**[2]

[1] ananth@isis.vanderbilt.edu,
[2] gabor.karsai@vanderbilt.edu
Institute for Software Integrated Systems,
Vanderbilt University
Nashville, TN 37203 USA

Abstract: Graph transformation is often used to transform domain models from one domain specific language (DSML) to another. In some cases, the DSMLs are based on a formalism that has many implementation variants, such as Statecharts. For instance, it could be necessary to transform iLogix Statechart models into Matlab Stateflow models. The preservation of behavior of the models is crucial in such transformations. Bisimulation has previously been demonstrated as an approach to verifying behavior preservation, and semantic anchoring is an approach to specifying the dynamic semantics of DSMLs. We propose a method to verify behavior preservation, using bisimulation in conjunction with semantic anchoring. We will consider two hypothetical variants of the Statecharts formalism, and specify the operational semantics of each variant by semantic anchoring, using Abstract State Machines as a common semantic framework. We then establish bisimulation properties to verify if the behavior models of the source and target Statechart models are equivalent for a particular execution of the transformation.

Keywords: Semantic Anchoring, Behavior Preservation, Bisimulation

1 Introduction

The preservation of the behavior of a model is crucial in many kinds of graph transformations. Consider a scenario where different users are exchanging information modeled through Statecharts. Since a formalism like Statecharts has many implementation variants [vdB94], it is often the case that the users use different variants. It is then necessary to transform the models from one variant to the other, such as from iLogix Statecharts to MATLAB Stateflow models. Such transformations could be accomplished through Graph Transformations. One related example is [ASK04], which uses graph transformations to translate Simulink/Stateflow models into Hybrid Automata. In all these cases, it is essential that the transformed model preserves the behavior of the source model.

Defining the behavior formally is a first step to verifying its preservation. Semantic Anchoring [CSAJ05] is a technique to specify the operational semantics of DSMLs using a semantic framework and anchoring rules. Bisimulation has been suggested as a method to check if a transformation preserves certain behavioral properties [NK06]. In this paper, we propose a method to verify behavior equivalence by using bisimulation in conjunction with semantic anchoring. As the practical Statecharts implementations vary in very subtle features that are not very suitable for a case study, we have devised two hypothetical variants with certain key differences that can better illustrate the complexities of the transformation. We will first specify their operational semantics by semantic anchoring, using Abstract State Machines as a common semantic framework. This will allow us to generate a behavior model for any instance model. We will then show how we can use bisimulation to check if the behavior models are equivalent. If the behavior models are equivalent, we can conclude that that particular instance of the transformation preserved the behavior correctly.

Though the source and target domains considered here are very similar, this approach can be applied to other types of transformations as long as the semantics of the source and the target languages can be represented using a common semantic framework such as Abstract State Machines.

2 Background

2.1 Statecharts

Statecharts [Har87] were first proposed by Harel to model the reactive behavior of systems. Statecharts were presented as an extension to conventional state machines, allowing hierarchy, concurrency and broadcast communication. Statecharts are constructed from *states* and *transitions*. States may be *simple* (basic states), *composite* (OR states) or *concurrent* (AND states). If a system is in a composite state, it is also in exactly one of its direct sub-states. If a system is in a concurrent state, it is also in all of its direct sub-states. A *state configuration* is a maximal set of states that the system can be active in simultaneously. Transitions take the system from one state configuration to another. Events are the basic units of broadcast communication. Transitions may be annotated with triggers, guards and actions. Triggers are the events required to activate the transition, actions are events that are broadcast as a result of taking the transition, and guards are boolean conditions that can enable or disable the transition.

Since the introduction of the Statecharts formalism, several variants have been proposed to overcome specific difficulties. A number of them have been surveyed in [vdB94]. One feature that the variants may differ on is whether inter-level transitions (which are transitions that cut across levels of hierarchy) are allowed or not. Another difference may be whether instantaneous states are permitted. Some such differences will be discussed in detail in the case study.

2.2 Semantic Anchoring

Domain Specific Modeling Languages (DSMLs) capture concepts, relationships and integrity constraints that will allow users to specify their systems declaratively. The meta-modeling step of designing a DSML specifies the syntax and static semantics of the DSML. Semantic anchoring [CSAJ05] concentrates on the specification of the dynamic semantics of the DSML. Semantic anchoring relies on the observation that a broad category of component behaviors can be represented by a small set of basic behavioral abstractions such as Finite State Machines, Timed Automata etc. We assume that the behavior of these abstractions are well understood and pre-

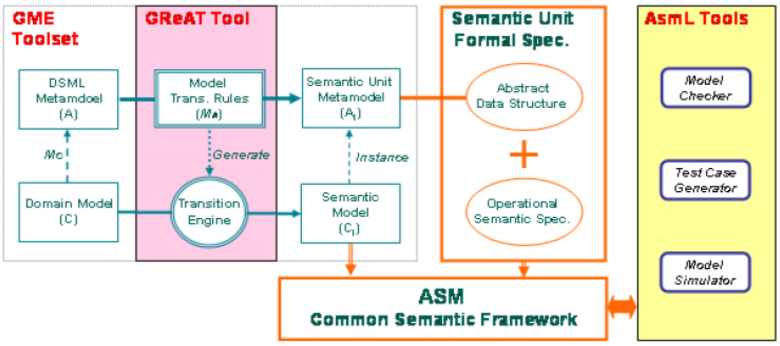


Figure 1: Tool Architecture for Semantic Anchoring

cisely defined. These basic abstractions are called *semantic units*. The behavioral semantics of the DSML is specified as a transformation from the DSML to the selected semantic unit. This last step is called the semantic anchoring. For instance, FSMs can be chosen as a semantic unit to represent the behavior of Statecharts.

The semantics of the behavioral abstraction can be specified using a mathematical model such as Abstract State Machines (ASMs). Microsoft's Abstract State Machine Language (AsmL) [Asm] provides a powerful language and associated set of tools for programming, simulating and model checking ASM models. The behavior model in the chosen behavioral abstraction, such as FSM, can be mapped to an AsmL Abstract Data Model, which can be simulated or checked using the available tools. The entire toolset is described in [Che06].

Figure 1 [CSAJ05] shows the tool architecture for specifying operational semantics to DSMLs through semantic anchoring. The GME [LBM+01] MIC toolset is used to define the static semantics and integrity constraints of the DSML, and to design the domain models. The semantic anchoring transformation is specified using GReAT [AKL03].

2.3 Bisimulation

Bisimulation [San04] is defined for Labeled Transitions Systems (LTS).
Given an LTS (S, Λ, →), where:
S is a set of states,
Λ is a set of labels,
and → is a set of transitions,
a relation R over S is a *bisimulation* if:

$$(p,q) \in R \text{ and } p \overset{\alpha}{\to} p' \text{ implies that there exists a } q' \in \mathrm{S} \text{ such that } q \overset{\alpha}{\to} q' \text{ and } (p',q') \in R,$$

and conversely,

$$q \overset{\alpha}{\to} q' \text{ implies that there exists a } p' \in \mathrm{S} \text{ such that } p \overset{\alpha}{\to} p' \text{ and } (p',q') \in R.$$

Bisimilarity is the union of all bisimulations. Bisimilarity is generally accepted as the finest form of behavioral equivalence. Note that while the definition is given in terms of a single set S,

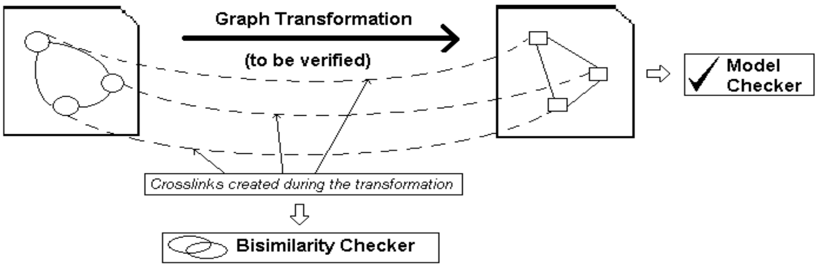


Figure 2: Architecture for verifying reachability preservation in a transformation

we can compare the equivalence of two transition systems by considering a global set containing both the systems' states.

We can also consider different "flavors" of bisimulation by varying the definition of transitions and labels. If we replace the transition $\rightarrow$ by a *weak transition* $\Rightarrow$, we get the definition of a *weak bisimulation* [HMS06]. Weak bisimulation allows us to compare the equivalence of systems at a level that is not as fine-grained as strict bisimilarity. Suitably defining what constitutes observable states and transitions, we can check if two transition systems have a weak bisimilarity. This will allow us to conclude whether the transition systems are behaviorally equivalent for all practical purposes, though there may not be a fine-grain equivalence. For instance, we may choose to ignore internal representations of data or intermediate states when comparing the transition systems for equivalence.

2.4 Verifying Instances of Graph Transformations

In an earlier work [NK06], we proposed a method for using bisimulation to verify each instance of a graph transformation. We believe that the strategy to verify each instance is less complex and more practical, as opposed to devising a correctness proof for the transformation itself, which may be intractable.

In this approach, we use bisimilarity to check if a reachability property is preserved by a certain instance of a transformation. Figure 2 shows the basic architecture used for the verification. The model transformation creates temporary associations between the source and target elements, which are used to trace the equivalence relation R. These links are then used by a bisimilarity checker, to check if this particular instance of input and output models preserve the same reachability behavior. If this check succeeds, the results of a model checker on the output model instance can be applied to the input model instance as well.

3 Verifying Behavior Preservation

Our approach to verifying behavior preservation relies on establishing a weak bisimilarity between the transition systems representing the behaviors of the source and target models. Figure 3 shows an overview of our approach. We would like to verify the graph transformation instance that transforms the *Variant 1* Statechart model *SC-1* into a *Variant 2* Statechart model *SC-2*. We generate the behavior models of both the source and target instances using semantic anchoring.

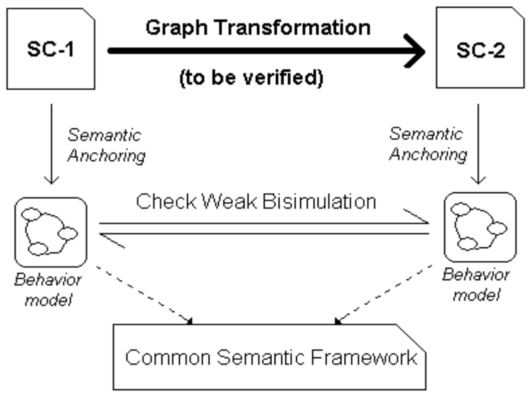


Figure 3: Framework for verifying behavior preservation using a common semantic framework and weak bisimilarity

We will use non-hierarchical FSMs as a semantic unit to represent the behavior of the source and target DSMLs. The semantic anchoring process will result in flat FSM representations of the source and target instances. We will then verify if there is a weak bisimulation based on some criteria, as explained in the following sub-sections.

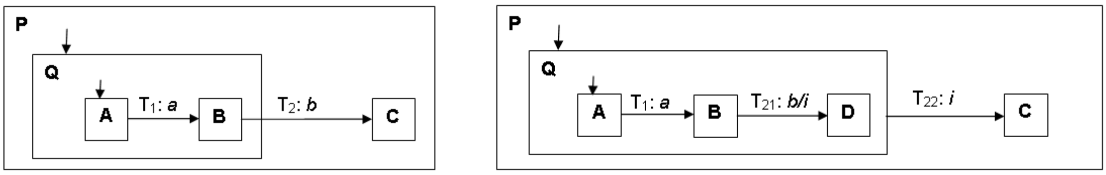
3.1 The Source and Target Languages

The popular Statecharts variants used currently vary in a number of subtle issues. For the sake of simplicity, we will use two hypothetical variants, called *Variant 1* and *Variant 2*, which differ in a small set of features that can be defined clearly. For this case study, we will only consider the synchronous time model. We will now list their features and their semantics.

3.1.1 Compositional Semantics

A language is said to have compositional semantics if the semantics of a compound object is completely defined by the semantics of its subcomponents. In other words, we do not have to look at the internal syntactical structure of the subcomponents. Such semantics are useful in verification. It is easy to see that compositional semantics is violated by allowing inter-level transitions and state references. Inter-level transitions are transitions that cut across levels of hierarchy. State references are a mechanism to allow the execution of a transition based on whether a certain parallel component is active, expressed as trigger conditions such as *in(State)*. In our case study, we will allow *Variant 1* to have inter-level transitions and state references, and not allow them in *Variant 2*.

Figure 4(a) shows a *Variant 1* Statechart, where transition T_2 is inter-level. Figure 4(b) shows a *Variant 2* Statechart, which tries to simulate the semantics of the *Variant 1* Statechart, by using a self-termination state to replace the inter-level transition. What happens if the transition T_2 in Figure 4(a) has an action *E*? In *Variant 1*, *E* will be active when the system enters state C, but in *Variant 2*, *E* will not be available when the system enters state C (since events are available only



(a) A sample *Variant 1* Statechart (b) A sample *Variant 2* Statechart

Figure 4: Sample Statechart models

in the step following the one in which they occur, and not in subsequent steps[1]). The semantics will be better reproduced if state *D* in Figure 4(b) is an *instantaneous state*. Instantaneous states will be discussed in the next subsection.

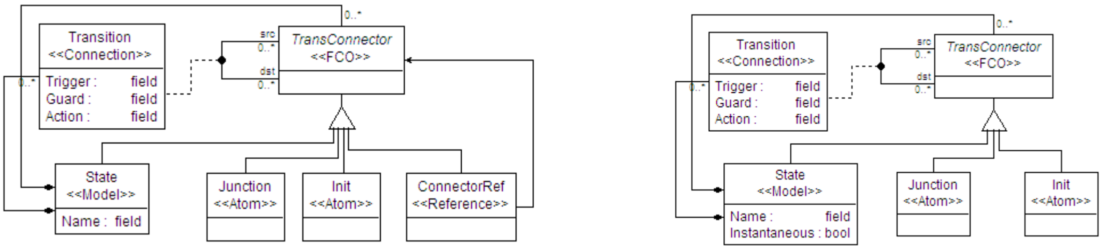
We must note that it may not be possible to represent all *Variant 1* Statecharts as *Variant 2* Statecharts. Our objective is not to find a *Variant 2* representation of any *Variant 1* Statechart. Rather, it is to verify whether a *Variant 2* Statechart generated by a graph transformation can be accepted as behaviorally equivalent to the *Variant 1* Statechart that was the input to the transformation.

3.1.2 Instantaneous States

An instantaneous state is a state that can be simultaneously entered and exited in a single step. Instantaneous states are not allowed in most common Statecharts variants. We will allow instantaneous states in *Variant 2* Statecharts, with the semantics that a step is not complete until there are no instantaneous states in the final state configuration of that step. The sequence of transitions leading to the final state configuration constitute a macro step, which is considered to be executed in zero time[2]. Events available at the start of the macro step are available throughout the macro step, and actions on any of the transitions in the macro step will be available in the step following the macro step. For providing finer semantics, we introduce the notion of instantaneous actions to *Variant 2* Statecharts, which will be available in the same step. In Figure 4(b), *D* will be an instantaneous state, and *i* will be an instantaneous action. The sequence T_{21}, T_{22} will form a macro step, with *B* as the starting state and *C* as the ending state. To an external observer, the instantaneous state *D* and instantaneous action *i* will not be visible, and the macro step will appear to be a transition whose triggers and actions will be the aggregate of the triggers and actions of the sequence of transitions. In other words, the macro step will be identical to a transition whose trigger is the conjunction of the triggers for each micro step, and all the trigger events must be enabled at the start of the macro step. The conjunction of all the non-instantaneous actions will be available for the step following the macro step. The effect of the macro step (T_{21}, T_{22}) in Figure 4(b) will be identical to transition T_2 in Figure 4(a).

[1] The durability of events is itself an issue that Statecharts variants may differ on, and is explained in [vdB94]

[2] For the purposes of this case study, we will not go into the issues of infinite sequences of transition executions.



(a) Statechart *Variant 1* (b) Statechart *Variant 2*

Figure 5: GME Meta-models for Statechart Variants

3.1.3 State References

Some Statecharts variants allow triggers to be specified by referencing the activity of other, parallel states. For instance, the condition *in(S)* is true when state *S* is active, and entering or exiting *S* will result in events *en(S)* and *ex(S)* respectively. We will allow state references in *Variant 1*, and not in *Variant 2*. The graph transformation that will generate the *Variant 2* Statechart will replace the events for *en(S)* and *ex(S)* when necessary, but will not handle the condition *in(S)*. The replaced events will be considered equivalent when verifying behavior preservation. *Variant 1* Statecharts which use the condition *in(S)* cannot be transformed by the graph transformation which we will consider in the case study (*in(S)* can be simulated by a system of self loops, but we chose not to implement it in the case study, to avoid overly complicating the transformation).

Figures 5(a) and 5(b) show the GME meta-models for the two Statechart variants for the case study. The GME modeling environment allows connections to only reside under a single parent. Thus, inter-level transitions must be represented by using references to states that are under another parent. The absence of a *ConnectionRef* prohibits inter-level transitions in Variant 2. States can be marked as instantaneous in Variant 2. The absence of state references in Variant 2 is enforced by the operational semantics defined by semantic anchoring for this variant.

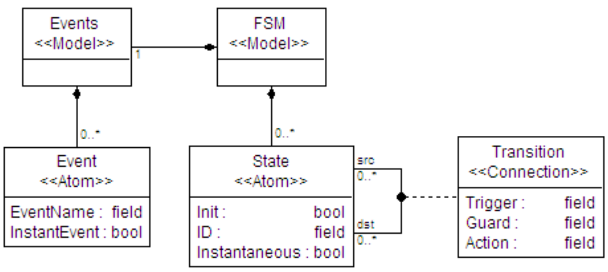
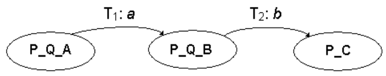
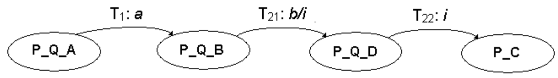


Figure 6: GME Meta-model for FSM semantic unit



(a) for *Variant 1* sample



(b) for *Variant 2* sample

Figure 7: FSM semantic models

3.2 Operational Semantics Using Semantic Anchoring

We will use a flat (non-hierarchical) FSM as the semantic unit to represent the behavior of the Statecharts variants. The meta-model of the semantic unit is shown in Figure 6. The FSM model allows instantaneous states and events, and can model the behavior of both the Statecharts variants discussed above. The semantics of this FSM semantic unit is defined in AsmL [Asm], the Abstract State Machine Language developed by Microsoft Research. This is done by specifying an Abstract Data Model in AsmL, corresponding to the constructs of the FSM semantic unit. For instance, *Events* are modeled as below:

```
interface Event
structure ModelEvent implements Event
structure LocalEvent implements Event
structure InstantEvent implements Event
```

ModelEvent models events input to the model, *LocalEvent* models the events generated in a step, and *InstantEvents* models events generated by instantaneous actions, which will be available in the same step.

The FSM itself, the states and the transitions are modeled using AsmL *class* constructs:

```
class FSM
id as String
var outputEvents as Seq of ModelEvent
var localEvents as Set of LocalEvent
...
class State
id as String
var active as Boolean = false
var instantaneous as Boolean
var outTransitions as Set of Transition
...
class Transition
...
```

The FSM behavior is modeled using operational rules on the data structures. The operational rules define the step semantics of the FSM instance's reaction to input events. These rules are called during the execution of the model in AsmL. The FSM instance's reaction is simulated by updating the data fields and activating the enabled transitions. The AsmL model of the instances can be obtained by instantiating the states and transitions based on the model. The details of creating the AsmL abstract model can be found in [CSAJ05].

The specification of the behavior of the Statecharts variants will be expressed via semantic anchoring, as a transformation from the Statechart model to the FSM semantic unit. Figure 7(a) shows the FSM behavior model of the Statechart in Figure 4(a). The first step in the transformation will to convert the hierarchical Statechart model into a non-hierarchical FSM model, by enumerating all possible state configurations. Each legal state configuration of the Statechart

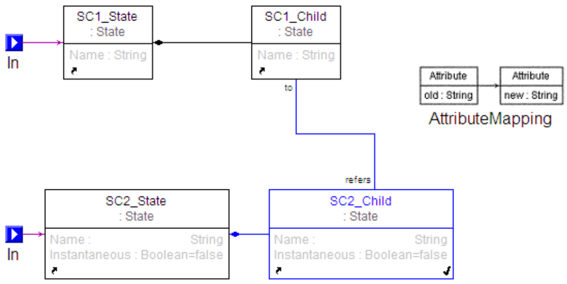


Figure 8: Sample GReAT rule

will be represented by a unique state in the FSM. For instance, the state *P_Q_A* in Figure 7(a) represents the state configuration consisting of the states *P*, *Q* and *A* in Figure 4(a). The next step is to transform the transitions. The transitions are extracted from the Statecharts model, and the source and target state configurations are determined. The corresponding unique states in the FSM are located, and a transition is constructed between them. The trigger, action and guards are then updated in the FSM model.

This transformation of the Statechart model into the FSM semantic unit is a behavior specification, and itself is not verified. We may use the technique described in [NK06] to verify the behavior model by bisimulation if necessary, but the behavior specification may be considered straightforward enough to not require further verification. We are mainly interested in verifying the transformation between the Statechart variants, and not the behavior specification of each variant.

3.3 Setting up the Transformation

We will now describe the graph transformation for our case study. This transformation will take a *Variant 1* Statechart and convert it into a *Variant 2* Statechart.

The main tasks of this transformation will be to convert inter-level transitions in *Variant 1* into normal transitions in *Variant 2* and to replace any state referencing actions by regular actions. Compositional semantics can be developed by replacing inter-level transitions with concepts called self-start and self-termination. In Figure 4(b), the state *D* can be thought of as a self-termination state, with the help of which the sequence of transitions T_{21}, T_{22} replace an interlevel transition. Similarly, self-start states can be used in the case of interlevel transitions entering a state and terminating in one of its substates.

In the transformation, every state in the input Statechart is first copied on to the output. Figure 8 shows a simple graph transformation rule in GReAT, where a new state is created in the *Variant 2* Statechart for a child state found in the *Variant 1* Statechart. The newly created elements (SC2_Child and the edge to SC1_Child in Figure 8) appear in blue color in the rule (when seen in color), with a tick mark in the bottom right. The *Attribute Mapping* block is a special construct in GReAT that allows us to perform additional functions, used in this case to set up the label of the state. We also track the equivalent states by creating a direct link between them. The transitions are extracted one by one, and if the source and destination states are contained in the same parent, a transition is created in the output between the corresponding source and destination states. If the transition is inter-level, then a self-start or self-termination state is added to the deeper of the two states, and the parent is marked as the source or target. The process is repeated until the source and target states are under the same parent. An automatic system of naming and numbering the intermediate instantaneous states and actions will ensure that each inter-level transition is reproduced uniquely.

If there are state references in the input model, they will be copied in the output model as normal events, using a special naming convention for identification. All such events will then be added as actions to all transitions into or out of the respective states being referenced. For instance, if a trigger *en(S)* is used in a *Variant 1* Statechart, it will be replaced by *en_S* in the output, and the action *en_S* will be added to all the transitions into state S in the output. The occurrence of *in(S)* in the input model will be flagged as an error.

3.4 Verifying Behavior Preservation

Our objective here is to verify if a certain instance of the transformation produced a *Variant 2* Statechart that preserved the behavior of the *Variant 1* Statechart given as input. To verify this, we get the semantic model of the input instance and the semantic model of the output instance, and check if they are behaviorally equivalent. This is done by establishing a weak bisimulation relation between the two behavior models.

3.4.1 Weak Bisimulation

According to the semantics we have described, we consider the two Statechart models in figures 4(a) and 4(b) to be behaviorally equivalent. But it is obvious that the two behavior models in Figures 7(a) and 7(b) are not bisimilar. We thus turn to a practically useful notion of bisimilarity that will help us in this scenario, the notion of *weak bisimilarity*.

In Figure 7(b), the state *P_Q_D* and the action i are instantaneous. To an external observer, the sequence of transitions T_{21}, T_{22} will appear identical to the transition T_2 in Figure 7(a). To the observer, the two systems are behaviorally equivalent, even though they may vary internally. Weak bisimulation allows us to weaken the notion of what constitutes a transition, allowing us to set the granularity at which we accept two systems as behaviorally equivalent. In our scenario, we argue that it is acceptable to consider only non-instantaneous states as states of the transition system, and a transition as one that goes from one non-instantaneous state to another, passing through any number of instantaneous states. We consider such transitions as a single *weak* transition, whose trigger and action are the aggregate of the series of transitions, disregarding all instantaneous states and actions.

We now rephrase our earlier definition of bisimulation, to define a weak bisimulation for FSM models. We define an equivalence relation R between the non-instantaneous states of two FSM models, for the current study, by simply stating that two states p and q are in R if they have the same label. Given this equivalence relation, we define a somewhat novel definition of weak bisimulation that is useful for our purpose:

$$\forall\ (p, q) \in R \text{ and } \forall \alpha\text{: } p \overset{\alpha}{\Rightarrow} p',\ \exists\ q' \text{ such that } q \overset{\alpha}{\Rightarrow} q' \text{ and } (p', q') \in R,$$

and conversely,

$$\forall \alpha \colon q \overset{\alpha}{\Rightarrow} q',\ \exists\, p' \text{ such that } p \overset{\alpha}{\Rightarrow} p' \text{ and } (p', q') \in R.$$

where p, q, p', q' are all non-instantaneous states, the transition $\Rightarrow$ is from one non-instantaneous state to another, and α is the aggregate of the events for the transition, disregarding instantaneous states and actions (The label α constitutes both the cause of a transition and its effect. In our implementation, we represent α as a comma separated list of the events that are the triggers and the actions of the transition. In the case of a weak transition, this list will include all the non-instantaneous events that are the triggers and the actions of the sequence of transitions which constitute the weak transition). According to this definition, the FSM models in Figures 7(a) and 7(b) are weakly bisimilar.

3.4.2 Checking for Weak Bisimulation

To verify weak bisimulation, we will first reduce the FSM into non-instantaneous states and weak transitions. This is done by tracing through the FSM model, and aggregating any sequence of transitions through instantaneous states. In this case, only *Variant 2* Statecharts will result in FSM models with instantaneous states, and the reduction step is not required for FSM models of *Variant 1* Statecharts.

The next step is to establish the equivalence relation R. For this study, we will consider two states p and q to be in R if they have the same label. Our transformation is set up in such a way that newly created states in the *Variant 2* Statechart model are labeled depending upon the *Variant 1* state they were created from. The FSM states are similarly labeled depending on the labels of the states in the Statechart state-configuration that they represent. We will consider that two states are equivalent if they represent the same configuration in the two Statecharts.

After these two steps, for each of the FSMs, we list all the states in a table, followed by all the outgoing transitions for each of these states. We then step through the states one by one, verifying that the weak bisimulation holds according to the definition above. After stepping through the list of states for both the FSMs, we can conclude whether the generated *Variant 2* Statechart is behaviorally equivalent to the input *Variant 1* Statechart.

4 Related Work

4.1 Graph Transformation Based Operational Semantics

[CHM00] and [Var04] present approaches to specify the operational semantics of DSMLs using graph transformations. [Var04] presents a meta-level analysis technique where the semantics of a modeling language are defined using graph transformation rules. A transition system is generated for each instance model, which can be verified using a model checker. This is an alternative to the semantic anchoring approach. We find the semantic anchoring approach easier to use, as we can choose a semantic unit such as non-hierarchical FSMs, and use that to verify weak bisimulation. But as long as the semantics of the input and output languages can be specified in terms of a common representation, and we define the equivalence relation and rules for weak bisimulation appropriately, behavior preservation can be verified.

4.2 Certifiable Program Generation

[DF05] considers the problem of verification of generated code by focusing on each individual generated program, instead of verifying the program generator itself. The generator is extended such that it produces all logical annotations that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the program does not violate certain conditions during its execution. While the proofs in this case are not related to semantic correctness, the idea of providing an instance level certificate of correctness instead of proving the correctness of the generator has been a great motivation for our ideas.

5 Conclusions and Future Work

Semantic anchoring is a useful method for formally specifying the dynamic semantics of DSMLs. We have shown here that it allows us to use bisimulation to verify behavior equivalence across a transformation. This technique is especially useful in cases where it is hard to compare the source and target languages directly. As long as their behavior can be specified using a common semantic framework, we can verify behavioral equivalence by using bisimulation. We have also shown that weak bisimulation is a practical way to determine acceptable behavioral equivalence. As in [NK06], we believe that it is more practical to verify whether an instance of a transformation succeeded in preserving behavior, instead of providing a proof of correctness for the transformation itself.

Further research in determining semantic units that can represent a wide variety of DSMLs will allow us to use this technique for a larger range of transformations. We may also consider representing a specific behavioral property using semantic anchoring, as opposed to the complete operational semantics of the language. This will allow us to check the preservation of a specific behavior in languages that are otherwise very different. The checking of weak bisimulation may also be refined to be more efficient.

Acknowledgements: This material is based upon work supported by the National Science Foundation under Grant No. 0509098. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Bibliography

[AKL03] A. Agrawal, G. Karsai, A. Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: 18th annual ACM SIGPLAN conference on OOP, systems, languages, and applications*. Pp. 8–15. ACM Press, New York, NY, USA, 2003.
doi:http://doi.acm.org/10.1145/949344.949347

[ASK04] A. Agrawal, G. Simon, G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electr. Notes Theor. Comput. Sci.* 109:43–56, 2004.

[Asm] The Abstract State Machine Language. http://www.research.microsoft.com/fse/asml.

[vdB94] M. von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Pp. 128–148. Springer-Verlag, London, UK, 1994.

[Che06] K. Chen. *A Semantic Anchoring Infrastructure for Model-Integrated Computing*. PhD thesis, Vanderbilt University, Nashville, TN 37203, Aug. 2006.

[CHM00] A. Corradini, R. Heckel, U. Montanari. Graphical Operational Semantics. In *ICALP Satellite Workshops*. Pp. 411–418. 2000.
citeseer.ist.psu.edu/corradini00graphical.html

[CSAJ05] K. Chen, J. Sztipanovits, S. Abdelwahed, E. K. Jackson. Semantic Anchoring with Model Transformations. In *ECMDA-FA*. Pp. 115–129. 2005.

[CSN05] K. Chen, J. Sztipanovits, S. Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. Pp. 35–43. ACM Press, New York, NY, USA, 2005.
doi:http://doi.acm.org/10.1145/1086228.1086236

[DF05] E. Denney, B. Fischer. Certifiable Program Generation. In *GPCE*. Pp. 17–28. 2005.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3):231–274, 1987.
doi:http://dx.doi.org/10.1016/0167-6423(87)90035-9

[HMS06] W. Harwood, F. Moller, A. Setzer. Weak Bisimulation Approximants. In *CSL'06: The 15th International Conference on Computer Science Logic*. Lecture Notes in Computer Science, Szeged, Hungary, 2006.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. *Computer* 34(11):44–51, 2001.
doi:http://dx.doi.org/10.1109/2.963443

[Mar92] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*. Pp. 550–564. Springer-Verlag, London, UK, 1992.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In Bruni and Varro (eds.), *Graph Transformation and Visual Modeling Techniques GT-VMT 2006*. Electronic Notes in Theoretical Computer Science, pp. 185–194. 2006.

[San04] D. Sangiorgi. Bisimulation: From The Origins to Today. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. Pp. 298–302. IEEE Computer Society, Washington, DC, USA, 2004. doi:http://dx.doi.org/10.1109/LICS.2004.13

[Var04] D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and Systems Modeling* 3(2):85–113, May 2004. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2004/sosym2004_varro.pdf